# 第 4 章　Verilog HDL

**本章概要**：本章介绍硬件描述语言 Verilog HDL 的语言规则、数据类型和语句结构，并介绍最基本、最典型的数字逻辑电路的 Verilog HDL 描述，作为 Verilog HDL 工程设计的基础。

**知识要点**：（1）Verilog HDL 设计模块的基本结构；

（2）Verilog HDL 的语言规则；

（3）用 Verilog HDL 实现各种类型电路及系统设计的方法；

（4）Verilog HDL 设计流程；

（5）Verilog HDL 的仿真。

**教学安排**：本章教学安排 **8 学时**。通过本章的学习，使读者熟悉 Verilog HDL 设计模块的基本结构和 Verilog HDL 的语言规则，进而掌握 Verilog HDL 的编程方法，并使读者在第 2 章学习的基础上，进一步掌握 EDA 技术的 Verilog HDL 文本输入设计法。

## 4.1　Verilog HDL 设计模块的基本结构

Verilog HDL 程序设计是由模块（module）构成的，设计模块的基本结构如图 4.1 所示。一个完整的 Verilog HDL 设计模块包括模块端口定义、I/O 声明、信号类型声明和功能描述 4 个部分。

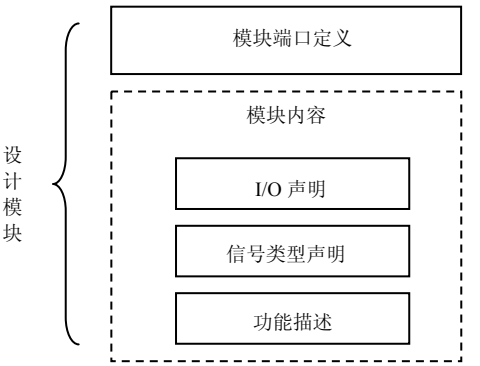


图 4.1　Verilog HDL 程序模块结构图

### 4.1.1　模块端口定义

模块端口定义用来声明设计模块名称及相应的（输入/输出）端口，端口定义格式如下：

```
module 模块名(端口 1,端口 2,端口 3, ……);
```

式中，module 是模块定义的关键词，模块名是设计电路的名称，它由用户按照标识符规则命名，也是保存的源文件名称。例如，设计一个 2 输入 4 与非门电路时，可以用 CT7400 作为模块名，并用 CT7400.v（.v 是 Verilog HDL 源文件的属性后缀）保存设计的源文件。在端口定义的圆括号

中，是设计电路与外界联系的全部输入/输出端口信号或引脚，它是设计模块对外的一个通信界面，是外界可以看到的部分（不包含电源和接地端），多个端口名之间用逗号“,”分隔。例如，用 adder1 作为 1 位全加器的 Verilog HDL 设计模块名，sum 是求和输出，cout 是向高位的进位输出，ina 和 inb 是两个加数的输入，cin 是低位进位输入，则 adder1 模块的端口定义为：

```
module adder1(sum,cout,a,b,cin);
```

### 4.1.2 模块内容

模块内容包括 I/O 声明、信号类型声明和功能描述。

**1．模块的 I/O 声明**

模块的 I/O 声明用来声明模块端口定义中各端口数据的流动方向，包括 input（输入）、output（输出）和 inout（双向）。I/O 声明格式如下：

```
input   端口 1,端口 2,端口 3,……;//声明输入端口
output  端口 1,端口 2,端口 3,……;//声明输出端口
```

例如，1 位全加器的 I/O 声明为：

```
input   a,b,cin;
output  sum,cout;
```

**2．变量类型声明**

变量类型声明用来声明设计电路的功能描述中所用的变量的数据类型和函数。变量的类型主要有 wire（连线）、reg（寄存器）、integer（整型）、real（实型）和 time（时间）等。例如：

```
wire    a,b,cin;          //声明 a,b,cin 是 wire 变量
reg     cout;             //声明 cout 是 reg 型变量
reg[7:0] q;               //声明 q 是 8 位 reg 型变量
```

在 Verilog HDL 的 2001 版本或以上版本，允许将 I/O 声明和变量类型放在一条语句中，例如：

```
output reg[7:0] q;        //声明 q 是 8 位 reg 型输出变量
```

关于变量声明部分内容将在后续的章节详细介绍。

**3．功能描述**

功能描述是 Verilog HDL 程序设计中最主要的部分，用来描述设计模块的内部结构和模块端口之间的逻辑关系，在电路上相当于器件的内部电路结构。功能描述可以用 assign 语句、元件例化（instantiate）、always 块语句、initial 块语句等方法来实现，通常把确定这些设计模块描述的方法称为建模。

（1）用 assign 语句建模

用 assign 语句建模的方法很简单，只需要在“assign”后面再加一个表达式即可。assign 语句一般适合对组合逻辑进行赋值，称为连续赋值方式。

**【例 4.1】**1 位全加器的设计。

1 位全加器的逻辑符号如图 4.2 所示，其中 sum 是全加器的和输出端，cout 是进位输出端，a 和 b 是两个加数输入端，cin 是低位进位输入端。

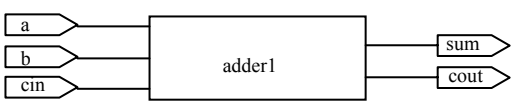


图 4.2　1 位全加器的逻辑符号

全加器设计电路的 Verilog HDL 源程序 adder1.v 如下：

```
module  adder1(sum,cout,a,b,cin);
input   a,b,cin;
```

```
output  sum,cout;
assign{cout,sum} = a+b+cin;
endmodule
```

在例 4.1 中，用语句“assign {cont,sum} = a+b+cin;”实现 1 位全加器的进位 cout 与和输出 sum 的建模。在语句表达式中，用并接运算符“{}”将 cont、sum 这两个 1 位操作数并接为一个 2 位操作数，位于左边的操作数的权值高，位于右边的操作数的权值低。

（2）用元件例化（instantiate）方式建模

元件例化方式建模是利用 Verilog HDL 提供的元件库实现的。例如，用与门例化元件定义一个三输入端与门可以写为

```
and myand3(y,a,b,c);
```

其中，and 是 Verilog HDL 元件库中“与门”元件名，myand3 是可选的例化名（相当于元件 and 插在印制电路板上的插座名），y 是与门的输出端，a、b 和 c 是输入端。完整的 3 输入端与门电路的 Verilog HDL 源程序 and_3.v 如下：

```
module  and_3(y,a,b,c);
input   a,b,c;
output  y;
and myand3(y,a,b,c);
endmodule
```

（3）用 always 块语句建模

always 块语句可以用于设计组合逻辑和时序逻辑，但时序逻辑必须用 always 块语句来建模。一个程序设计模块中，可以包含一个或多个 always 块语句。程序运行中，若 always 块语句的敏感参数发生变化，就执行一遍 always 块中的语句，产生新的结果。

**【例 4.2】**4 位十进制加法计数器的设计。

用 Verilog HDL 设计的 4 位十进制加法计数器的元件符号如图 4.3 所示，其中 q 是 4 位十进制加法计数器的输出端（4 位向量）；cout 是进位输出端（1 位）；clk 是时钟控制输入端，上升沿为有效边沿；clrn 是同步复位输入端，低电平有效，当 clrn 的下降沿到来时且 clrn=0，则计数器被复位，q=0。

4 位十进制加法计数器的 Verilog HDL 源程序 cnt10.v 如下：

```
module cnt10(clk,clrn,q,cout);
  input     clk,clrn;
  output reg[3:0] q;
  output reg    cout;
always  @(posedge clk or negedge clrn)
  begin
if(~clrn)q = 0;
    else begin
        if (q==9)q = 0;
    else q = q+1;
    if(q==9)cout = 1;
    else cout = 0;  end
end
endmodule
```

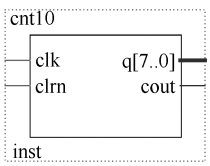


图 4.3　4 位十进制加法计数器的逻辑符号

在例 4.2 的源程序中，用 always 块语句来实现 8 位二进制加法计数器的建模。@(posedge clk or negedge clrn)是时间控制敏感函数，表示 clk 上升沿到来或者 clrn 下降沿到来的敏感时刻，

alwyas 块中的全部语句就执行一遍。另外，在程序的最后用“if (q==9)cout=1;else cout=0;”两条语句产生进位输出。4 位十进制加法计数器 cnt10 的仿真结果如图 4.4 所示。

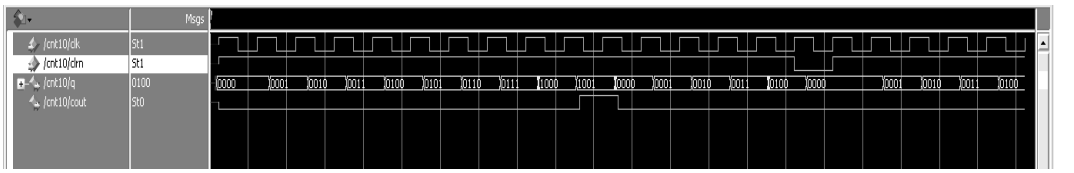


图 4.4　4 位十进制加法计数器 cnt10 的仿真结果图

（4）用 initial 块语句建模

initial 块语句与 always 块语句类似，不过在程序中它只执行 1 次就结束了。initial 块语句主要用于设计电路的初始化操作和仿真。例如在电子日历的设计中，可以将日历的日期初始化为 2000（年）01（月）01（日）。

从例 4.1 和例 4.2 可以看出 Verilog HDL 程序设计模块的基本结构。

① Verilog HDL 程序是由模块构成的。每个模块的内容都是嵌在 module 和 endmodule 语句之间，每个模块实现特定的功能，模块是可以进行层次嵌套的。

② 每个模块首先要进行端口定义，并声明输入（input）、输出（output）或双向（inout），然后对模块的功能进行逻辑描述。

③ Verilog HDL 程序的书写格式自由，一行可以书写一条语句或多条语句，一条语句也可以分为多行书写。

④ 除了 end 或以 end 开头的关键词（如 endmodule）语句外，每条语句后必须要有分号“;”。程序中的关键词全部用小写字母书写，标点符号全部用半角符号。

⑤ 可以用/*……*/或//……对 Verilog HDL 程序的任何部分作注释。一个完整的源程序都应当加上必要的注释，以加强程序的可读性。

## 4.2　Verilog HDL 的词法

Verilog HDL 源程序由空白符号分隔的词法符号流组成。词法符号包括空白符、注释、操作符、常数、字符串、标识符和关键词。准确无误地理解和掌握 Verilog HDL 的词法规则和用法，对正确完成 Verilog HDL 程序设计十分重要。

### 4.2.1　空白符和注释

Verilog HDL 的空白符包括空格、tab 符号、换行和换页。空白符用来分隔各种不同的词法符号，合理地使用空白符可以使源程序具有一定的可读性和编程风格。空白符如果不是出现在字符串中，编译源程序时将被忽略。

注释分为行注释和块注释两种方式。行注释用符号//（两个斜杠）开始，注释到本行结束。块注释用/*开始，用*/结束。块注释可以跨越多行，但它们不能嵌套。

### 4.2.2　常数

Verilog HDL 中的常数包括 3 种：数字、未知 x 和高阻 z。数字可以用二进制、十进制、八进制和十六进制等 4 种不同的数制来表示，完整的数字格式为：

```
<位宽>'<进制符号><数字>
```

其中，位宽表示数字对应的二进制数的位数宽度；进制符号包括 b 或 B（表示二进制数）、d 或 D（表示十进制数）、h 或 H（表示十六进制数）、o 或 O（表示八进制数）。例如：

```
8'b10110001          //表示位宽为 8 位的二进制数 10110001
8'hf5                //表示位宽为 8 位的十六进制数 f5
```

数字的位宽可以缺省，例如：

```
'b10110001           //表示二进制数
'hf5                 //表示十六进制数
```

十进制数的位宽和进制符号都可以缺省，例如：

```
125                  //表示十进制数 125
```

另外，用 x（或 X）和 z（或 Z）分别表示未知值和高阻值，它们可以出现在除了十进制数以外的数字形式中。x 和 z 的位数由所在的数字格式决定。在二进制数格式中，一个 x 或 z 表示 1 位未知位或 1 位高阻位；在十六进制数中，一个 x 或 z 表示 4 位未知位或 4 位高阻位；在八进制数中，一个 x 或 z 表示 3 位未知位或 3 位高阻位。例如：

```
'b1111xxxx      //等价'hfx
'b1101zzzz      //等价'hdz
```

### 4.2.3 字符串

字符串是用双引号括起来的可打印字符序列，它必须包含在同一行中。例如，"ABC"，"A BOY. "，"A"，"1234"都是字符串。

### 4.2.4 关键词

关键词（也称关键字）是 Verilog HDL 预先定义的单词或单词的组合，它们在程序中有不同的使用目的。例如，module 和 endmodule 用来指出源程序模块的开始和结束；assign 用来描述一个逻辑表达式等。Verilog-1995 的关键词有 97 个，Verilog-2001 增加了 5 个，共 102 个，详见表 4.1，每个关键词全部由小写字母组成，少数关键词中包含“0”或“1”数字。

### 4.2.5 标识符

标识符是用户编程时为常量、变量、模块、寄存器、端口、连线、示例和 begin-end 块等元素定义的名称。标识符可以是字母、数字、下画线“_”等符号组成的任意序列。定义标识符时应遵循如下规则：

① 首字符不能是数字；

② 字符数不能多于 1024 个；

③ 大、小写字母是不同的（仅限于 Verilog-1995 版本）；

④ 不能与关键词同名。

例如，a、b、adder、adder8、name_adder 都是正确的标识符；而 1a、?b 是错误的标识符。

与 VHDL'93 标准支持扩展标识符类似，Verilog HDL 允许使用转义标识符。转义标识符中可以包含任意的可打印字符，转义标识符从空白符号开始，以反斜杠“\”作为开始标记，到下一个空白符号结束，反斜杠不是标识符的一部分。下面是转义标识符的示例：

```
\74LS00
\a+b
```

### 4.2.6 操作符

操作符也称为运算符，是 Verilog HDL 预定义的函数名字，这些函数对被操作的对象（即操作数）进行规定的运算，得到一个结果。操作符通常由 1～3 个字符组成，例如，“+”表示加操作，“==”（2 个=字符）表示逻辑等操作，“===”（3 个=字符）表示全等操作。有些操作符的操作数只有 1 个，称为单目操作；有些操作符的操作数有 2 个，称为双目操作；有些操作符的操作数有 3 个，称为三目操作。

**表 4.1　Verilog HDL 的关键词**

| always | and | assign | begin | buf |
|---|---|---|---|---|
| bufi0 | bufu1 | case | casex | casez |
| cmos | deassign | default | defparam | disable |
| edge | else | end | endcase | endfunction |
| endmodule | endprimitive | endspecify | endtable | endtask |
| event | for | force | forever | fork |
| function | highz0 | highz1 | if | initial |
| inout | input | integer | join | large |
| macromodule | medium | module | nand | negedge |
| nmos | nor | not | notif0 | nottif1 |
| or | output | pmos | posedge | primitive |
| pull0 | pull1 | pulldown | pullup | remos |
| reg | release | Repeatr | rnmos | rpmos |
| rtran | rtranif0 | reranif1 | scalared | small |
| specify | specparam | strong0 | strong1 | supply0 |
| supply1 | table | task | time | tran |
| tranif0 | tranif1 | tri | tri0 | tri1 |
| triand | trior | vectored | wait | wand |
| weak0 | weak1 | while | wire | wor |
| xnot | xor | | | |

Verilog HDL 的操作符分为算术操作符、逻辑操作符、位运算、关系操作符、等值操作符、缩减操作符、转移操作符、条件操作符和并接操作符，共 9 类。

1．算术操作符（Arithmetic operators）

常用的算术操作符有 6 种：+（加）、-（减）、*（乘）、/（除）、%（求余）和**（乘方）。其中%是求余操作符，在两个整数相除的基础上，取出其余数。例如，5 % 6 的值为 5；13 % 5 的值是 3。整除（/）和求余（%）运算符可以方便电路的设计，如将二进制数转换为十进制数（8421BCD 码），但这两种运算符在综合过程中占用很多逻辑单元（LEs），所以一般电路设计最好不要使用。

2．逻辑操作符（Logical operators）

逻辑操作符包括：&&（逻辑与）、||（逻辑或）、!（逻辑非）。例如，A && B 表示 A 和 B 进行逻辑与运算；A || B 表示 A 和 B 进行逻辑或运算；!A 表示对 A 进行逻辑非运算。

3．位运算（Bitwise operators）

位运算是将两个操作数按对应位进行逻辑操作。位运算操作符包括：~（按位取反）、&（按位与）、|（按位或）、^（按位异或）、^~或~^（按位同或）。例如，设 A='b11010001，B='b00011001，则：

```
~A = 'b00101110
A & B = 'b00010001
A | B = 'b11011001
A ^ B = 'b11001000
A ^~ B = 'b00110111
```

在进行位运算时，当两个操作数的位宽不同时，计算机会自动将两个操作数按右端对齐，位数少的操作数会在高位用 0 补齐。

位运算与逻辑操作符运算的结果是相同的，因此，逻辑操作运算直接可以用位运算替代，例如，A && B 可以写成 A & B。

4．关系操作符（Relational operators）

关系操作符用来对两个操作数进行比较。关系操作符有： <（小于）、<=（小于等于）、>（大于）、>=（大于等于）。其中，<=也是赋值运算的赋值符号。

关系运算的结果是 1 位逻辑值。在进行关系运算时，如果关系是真，则计算结果为 1；如果关系是假，则计算结果为 0；如果某个操作数的值不定，则计算结果不定（未知 x），表示结果是模糊的。

5．等值操作符（Equality operators）

等值操作符包括 4 种：==（等于）、!=（不等于）、===（全等）、!==（不全等）。

等值运算的结果也是 1 位逻辑值，当运算结果为真时，返回值 1；为假则返回值 0。相等操作符（==）与全等操作符（===）的区别是：当进行相等运算时，两个操作数必须逐位相等，其比较结果的值才为 1（真），如果某些位是不定或高阻状态，其相等比较的结果就会是不定值；而进行全等运算时，对不定或高阻状态位也进行比较，当两个操作数完全一致时，其结果的值才为 1（真），否则结果为 0（假）。

例如，设 A = 'b1101xx01，B = 'b1101xx01，则：

```
A == B          运算的结果为 x（未知）
A === B         运算的结果为 1（真）
```

6．缩减操作符（Reduction operators）

缩减操作符包括：&（与）、~&（与非）、|（或）、~|（或非）、^（异或）、^~或~^（同或）。缩减操作运算法则与逻辑运算操作相同，但操作的运算对象只有一个。在进行缩减操作运算时，对操作数进行与、与非、或、或非、异或、同或等缩减操作运算，运算结果有 1 位“1”或“0”。例如，设 A = 'b11010001，则&A = 0（在与缩减运算中，只有 A 中的数字全为 1 时，结果才为 1）；|A = 1（在或缩减运算中，只有 A 中的数字全为 0 时，结果才为 0）。缩减操作相当于一个逻辑门，与缩减运算相当于一个与门，只有与门的全部输入为“1”时，输出（1 位）才为“1”，否则输出为“0”。

### 7．转移操作符（Shift operators）

转移操作符包括：>>（右移）、<<（左移）。其使用方法为：

```
操作数 >> n;      //将操作数的内容右移 n 位，同时从左边开始用 0 来填补移出的位数
操作数 << n;      //将操作数的内容左移 n 位，同时从右边开始用 0 来填补移出的位数
```

例如，设 A = 'b11010001，则 A >> 4 的结果是 A = 'b00001101；而 A << 4 的结果是 A = 'b00010000。

### 8. 条件操作符(Conditional operators)

条件操作符为：?:

条件操作符的操作数有 3 个，其使用格式为：

```
操作数 = 条件 ? 表达式 1:表达式 2;
```

即当条件为真（条件结果值为 1）时，操作数 = 表达式 1；为假（条件结果值为 0）时，操作数 = 表达式 2。

**【例 4.3】**用 Verilog HDL 的条件操作符设计三态输出电路。

三态输出电路如图 4.5 所示，其中 a 是 1 位数据输入端，f 是 1 位数据输出端，en 是使能控制输入端，高电平有效。当 en=1 时，电路工作，输出 f=a，当 en=0 时，电路不工作，输出为高阻态（f='bz）。

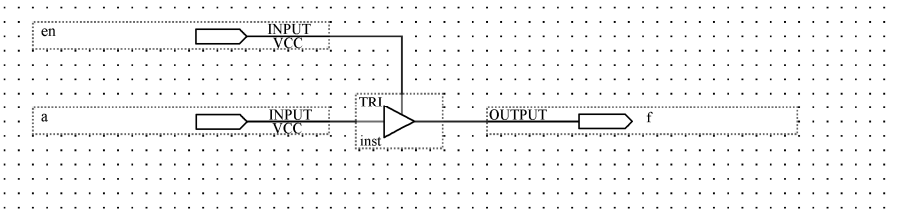


图 4.5　例 4.3 的硬件实现电路

用 Verilog HDL 的条件操作符设计三态输出电路的源程序如下：

```
module  tri_v(f,a,en);
input   a,en;
output  f;
assign  f = en ? a: 'bz;
endmodule
```

在本例中用了一个含有“?:”条件操作符的“assign f = en ? a: 'bz;”语句来描述三态输出电路，assign 语句的条件是变量 en，其值只有 0 和 1 两种。如果 en 为 1（真）时 y=a，为 0（假）时则 y='bz（高阻）。三态输出电路的仿真波形如图 4.6 所示（为了读者清楚阅图，本章的仿真波形是采用 Quartus II 9.0 软件的仿真工具 Waveform Editor 或 Quartus II 13.0 的 university program vwf 得到），在波形图中，当 en=0 时输出 f 为高阻态（高阻态是以在高低电平中部的粗线表示），当 en=1 时，输出 f 与输入 a 的波形相同。

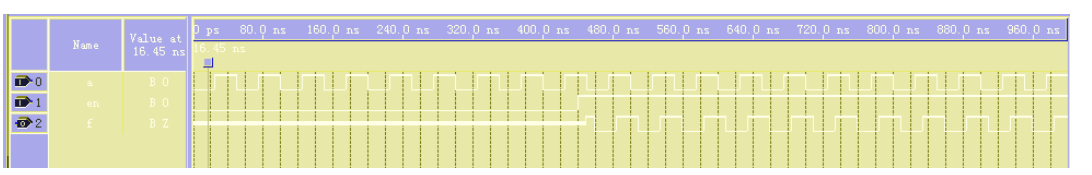


图 4.6　三态输出电路的仿真波形

9．并接操作符（Concatenation operators）

并接操作符为：{ }

并接操作符的使用格式为：

```
{操作数 1 的某些位,操作数 2 的某些位,……,操作数 n 的某些位};
```

即将操作数 1 的某些位与操作数 2 的某些位……与操作数 n 的某些位并接在一起，构成一个由这些数组成的多位数。例如，将 1 位全加器进位 cont 与和 sum 并接在一起使用，它们的结果由两个加数 a、b 及低位进位 cin 相加决定的表达式为：

```
{cont,sum}= a+b+cin;
```

10．操作符的优先级

操作符的优先级见表 4.2。表中顶部的操作符优先级最高，底部的最低，列在同一行的操作符的优先级相同。所有的操作符（?:操作符除外）在表达式中都是从左向右结合的。圆括号可以用来改变优先级，并使运算顺序更清晰，对操作符的优先级不能确定时，最好使用圆括号来确定表达式的优先顺序，既可以避免出错，也可以增加程序的可读性。

表 4.2　操作符的优先级

| 优先级序号 | 操　作　符 | 操作符名称 |
|---|---|---|
| 1 | ！、~ | 逻辑非、按位取反 |
| 2 | * 、/、 % | 乘、除、求余 |
| 3 | +、- | 加、减 |
| 4 | <<、>> | 左移、右移 |
| 5 | <、<=、>、>= | 小于、小于等于、大于、大于等于 |
| 6 | ==、!=、===、!== | 等于、不等于、全等、不全等 |
| 7 | &、~& | 缩减与、缩减与非 |
| 8 | ^、^~ | 缩减异或、缩减同或 |
| 9 | \|、~\| | 缩减或、缩减或非 |
| 10 | && | 逻辑与 |
| 11 | \|\| | 逻辑或 |
| 12 | ?: | 条件操作符 |

## 4.2.7　Verilog HDL 数据对象

Verilog HDL 数据对象是指用来存放各种类型数据的容器，包括常量和变量。

1．常量

常量是一个恒定不变的数值，一般在程序前部定义。常量定义格式为：

```
parameter 常量名 1 = 表达式,常量名 2 = 表达式, ……,常量名 n = 表达式;
```

其中，parameter 是常量定义关键词，常量名是用户定义的标识符，表达式是为常量赋的值。例如：

```
parameter Vcc = 5, fbus = 'b11010001;
```

上述语句定义了常量 Vcc 的值为十进制数 5，常量 fbus 的值为二进制数 11010001。

2．变量

变量是在程序运行时其值可以改变的量。在 Verilog HDL 中，变量分为网络型（nets type）和寄存器型（register type）两种。

（1）网络型变量（nets type）

nets 型变量是输出值始终根据输入变化而更新的变量，它一般用来定义硬件电路中的各种物理连线。Verilog HDL 提供了多种 nets 型变量，见表 4.3。

**表 4.3　常用的 nets 型变量及说明**

| 类　型 | 功 能 说 明 |
|---|---|
| wire、tri | 连线类型（两者功能完全相同） |
| wor、trior | 具有线或特性的连线（两者功能一致） |
| wand、triand | 具有线与特性的连线（两者功能一致） |
| tri1、tri0 | 分别为上拉电阻和下拉电阻 |
| supply1、supply0 | 分别为电源（逻辑 1）和地（逻辑 0） |

在 nets 型变量中，wire 型变量是最常用的一种。wire 型变量常用来表示以 assign 语句赋值的组合逻辑信号。Verilog HDL 模块中的输入/输出信号类型默认时自动定义为 wire 型。wire 型信号可以作为任何方程式的输入，也可以作为 assign 语句和例化元件的输出。对综合而言，wire 型变量的取值可以是 0、1、x 和 z。

wire 型变量的定义格式如下：

```
wire [位宽] 变量名 1,变量名 2,……,变量名 n;//位宽用于定义变量的二进制位数
```

例如：

```
wire        a,b,c;          //定义了 3 个 wire 型的变量，位宽均为 1 位（默认）
wire[7,0]   databus;        //定义了 1 个 wire 型的数据总线，位宽为 8 位
wire[15,0]  addrbus;        //定义了 1 个 wire 型的地址总线，位宽为 16 位
```

（2）寄存器型变量（register type）

register 型变量类似 VHDL 中的信号（Signal），是用来描述硬件系统的基本数据对象。它作为一种数值容器，不仅可以容纳当前值，也可以保持历史值，这一属性与触发器或寄存器的记忆功能有很好的对应关系。变量也是一种连接线，可以作为设计模块中各器件之间的信息传送通道。register 型变量与 wire 型变量的根本区别在于：register 型变量需要被明确地赋值，并且在被重新赋值前一直保持原值。wire 型变量在 assign 语句和元件例化语句中赋值，而 register 型变量是在 always、initial 等过程语句中定义，并通过过程语句赋值。

Verilog HDL 中的 register 型变量有 4 种，见表 4.4。integer、real 和 time 3 种寄存器型变量都是纯数学的抽象描述（不可综合），不对应任何具体的硬件电路，但它们可以描述与模拟有关的计算。例如，可以利用 time 型变量控制经过特定的时间后关闭显示等。

reg 型变量是数字系统中存储设备的抽象，常用于具体的硬件描述，因此是最常用的寄存器型变量，下面重点介绍 reg 型变量。

**表 4.4　常用的 register 型变量及说明**

| 类　型 | 功 能 说 明 |
|---|---|
| reg | 常用的寄存器型变量 |
| integer | 32 位带符号整数型变量 |
| real | 64 位带符号实数型变量 |
| time | 无符号时间型变量 |

reg 型变量定义的关键词是 reg，定义格式如下：

```
reg [位宽]  变量 1,变量 2,……,变量 n;
```

用 reg 定义的变量有一个范围选项（即位宽），默认的位宽是 1。位宽为 1 位的变量称为标量，位宽超过 1 位的变量称为向量。标量的定义不需要加位宽选项，例如：

```
reg a,b;                    //定义两个 reg 型变量 a，b
```

向量定义时需要位宽选项，例如：

```
reg[7:0]    data;        //定义 1 个 8 位寄存器型变量，最高有效位是 7，最低有效位是 0
reg[0:7]    data;        //定义 1 个 8 位寄存器型变量，最高有效位是 0，最低有效位是 7
```

向量定义后可以采用多种使用形式（即赋值）。

① 为整个向量赋值的形式为：

```
data='b00000000;
```

② 为向量的部分位赋值的形式为：

```
data[5:3]='b111;        //将 data 的第 5，4，3 位赋值为“111”
```

③ 为向量的某一位赋值的形式为：

```
data[7]=1;
```

（3）数组

若干个相同宽度的向量构成数组。在数字系统中，reg 型数组变量即为 memory（存储器）型变量。存储器型可以用如下语句定义：

```
reg[7:0]        mymemory[1023:0];
```

上述语句定义了一个 1024 个字存储器变量 mymemory，每个字的字长为 8 位。在表达式中可以用下面的语句来使用存储器：

```
mymemory[7] = 75;        //存储器 mymemory 的第 7 个字被赋值 75
assign A= mymemory[7]; //将存储器 mymemory 的第 7 个字的值赋给变量 A（存储器读）
```

## 4.3 Verilog HDL 的语句

语句是构成 Verilog HDL 程序不可缺少的部分。Verilog HDL 的语句包括赋值语句、条件语句、循环语句、结构声明语句和编译预处理语句等类型，每类语句又包括几种不同的语句。在这些语句中，有些语句属于顺序执行语句，有些语句属于并行执行语句。

### 4.3.1 赋值语句

在 Verilog HDL 中，赋值语句常用于描述硬件设计电路输出与输入之间的信息传送，改变输出结果。Verilog HDL 有门基元、连续赋值、过程赋值和非阻塞赋值 4 种赋值方法（即语句）。不同的赋值语句使输出产生新值的方法不同。

**1．门基元赋值语句**

门基元赋值语句的格式为：

```
基本逻辑门关键词        (门输出,门输入 1,门输入 2,……,门输入 n);
```

其中，基本逻辑门关键词是 Verilog HDL 预定义的逻辑门，包括 and、or、not、xor、nand、nor 等；圆括号中的内容是被描述门的输出和输入信号。例如，具有 4 个输入 a、b、c、d 和 y 输出的与非门，其门基元赋值语句为：

```
nand     (y,a,b,c,d);    //该语句与 assign y = ~(a & b & c & d);语句等效
```

**2．连续赋值语句**

连续赋值语句的关键词是 assign，赋值符号是“=”，赋值语句的格式为：

```
assign  赋值变量 = 表达式;
```

例如，具有 a、b、c、d 4 个输入和 y 为输出与非门的连续赋值语句为：

```
assign y = ~(a & b & c & d);
```

连续赋值语句的“=”号两边的变量都应该是 wire 型变量。在执行中，输出 y 的变化跟随

输入 a、b、c、d 的变化而变化，反映了信息传送的连续性。连续赋值语句用于逻辑门和组合逻辑电路的描述。

【例 4.4】4 输入端与非门的 Verilog HDL 源程序。

```
module  example_4(y,a,b,c,d);
output  y;
input   a,b,c,d;
assign  #1 y = ~(a&b&c&d);
endmodule
```

程序中的“#1”表示该门的输出与输入信号之间具有 1 个单位的时间延迟。

### 3．过程赋值语句

过程赋值语句出现在 initial 和 always 块语句中，赋值符号是“=”，它是顺序语句，语句格式为：

```
赋值变量 = 表达式;
```

在过程赋值语句中，赋值号“=”左边的赋值变量必须是 reg（寄存器）型变量，其值在该语句结束即可得到。如果一个块语句中包含若干条过程赋值语句，那么这些过程赋值语句按照语句编写的顺序由上至下一条一条地执行，前面的语句没有完成，后面的语句就不能执行，就如同被阻塞了一样。因此，过程赋值语句也称为阻塞赋值语句。

### 4．非阻塞赋值语句

非阻塞赋值语句也是出现在 initial 和 always 块语句中，赋值符号是“<=”，语句格式为：

```
赋值变量 <= 表达式;
```

在非阻塞赋值语句中，赋值号“<=”左边的赋值变量也必须是 reg 型变量，其值不像过程赋值语句那样在语句结束时即刻得到，而在该块语句结束才可得到。例如，在下面的块语句中包含 4 条赋值语句：

```
always  @(posedge clock)
m = 3;
n = 75;
n <= m;
r = n;
```

语句执行结束后，r 的值是 75，而不是 3，因为第 3 行是非阻塞赋值语句“n <= m”，该语句要等到本块语句结束时，n 的值才能改变。块语句中的“@(posedge clock)”是定时控制敏感函数，表示时钟信号 clock 的上升沿到来的敏感时刻。

过程赋值语句和非阻塞赋值语句都是在 initial 和 always 块语句中使用的，因此都称为过程赋值语句，只是赋值方式不同。过程赋值语句常用于数字系统的触发器、移位寄存器、计数器等时序逻辑电路的描述，也可用于组合逻辑电路的描述。

【例 4.5】上升沿触发的 D 触发器的 Verilog HDL 的源程序。

```
module  D_FF(q,d,clock);
input   d,clock;
output  q;
reg     q;
always  @(posedge clock)
q = d;
endmodule
```

在源程序中，q 是触发器的输出，属于 reg 型变量；d 和 clock 是输入，属于 wire 型变量

（由隐含规则定义）。在 always 块语句中，“posedge clock”是敏感变量，只有 clock 的正边沿（上升沿）到来时，D 触发器的输出 q=d，否则触发器的状态不变（处于保持状态）。

### 4.3.2 条件语句

条件语句包含 if 语句和 case 语句，它们都是顺序语句，应放在 always 或 initial 块语句中。

**1．if 语句**

完整的 Verilog HDL 的 if 语句结构如下：

```
if (表达式) begin 语句; end
else if (表达式)
  begin 语句; end
else
  begin 语句; end
```

根据需要，if 语句可以写为另外两种变化形式。

（1）if (表达式)

```
begin 语句; end
```

（2）if (表达式)

```
    begin 语句; end
else
    begin 语句; end
```

在 if 语句中，“表达式”一般为逻辑表达式或关系表达式，也可以是位宽为 1 位的变量。系统对表达式的值进行判断，若为 0，x，z，按“假”处理；若为 1，按“真”处理，执行指定的语句。语句可以是多句，多句时用“begin-end”语句括起来；也可以是单句，单句可以省略“begin-end”语句。对于 if 语句嵌套，如果不清楚 if 和 else 的匹配，最好用“begin-end”语句括起来。

if 语句及其变化形式属于条件语句，在程序中用来改变控制流程。

**【例 4.6】**8 线-3 线优先编码器的设计。

8 线-3 线优先编码器的功能见表 4.5，a0～a7 是 8 个信号输入端，a7 的优先级最高，a0 的优先级最低。当 a7 有效时（低电平 0），其他输入信号无效，编码输出 y2y1y0=111（a7 输入的编码）；如果 a7 无效（高电平 1），而 a6 有效，则 y2y1y0=110（a6 输入的编码）；以此类推。在传统的电路设计中，优先编码器的设计是一个相对困难的课题，而采用 Verilog HDL 的 if 语句，此类难题迎刃而解，充分体现了硬件描述语言在数字电路设计方面的优越性。

**表 4.5　8 线-3 线优先编码器的功能表**

| 输　入 | | | | | | | | 输　出 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| a0 | a1 | a2 | a3 | a4 | a5 | a6 | a7 | y2 | y1 | y0 |
| × | × | × | × | × | × | × | 0 | 1 | 1 | 1 |
| × | × | × | × | × | × | 0 | 1 | 1 | 1 | 0 |
| × | × | × | × | × | 0 | 1 | 1 | 1 | 0 | 1 |
| × | × | × | × | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| × | × | × | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| × | × | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| × | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

8 线-3 线优先编码器设计电路的 Verilog HDL 源程序 coder_8.v 如下：

```
module  coder_8(y,a);
```

```
input[7:0]      a;
output[2:0] y;
reg[2:0]        y;
always  @(a)
    begin
        if(~a[7])       y='b111;
        else if(~a[6]) y='b110;
        else if(~a[5]) y='b101;
        else if(~a[4]) y='b100;
        else if(~a[3]) y='b011;
        else if(~a[2]) y='b010;
        else if(~a[1]) y='b001;
        else if(~a[0])  y='b000;
        else            y='b000;
    end
endmodule
```

### 2．case 语句

case 语句是一种多分支的条件语句，完整的 case 语句的格式为：

```
case (表达式)
      选择值 1:       语句 1;
      选择值 2:       语句 2;
            ……          ;
      选择值 n:       语句 n;
      default:        语句 n+1;
  endcase
```

执行 case 语句时，首先计算表达式的值，然后执行在条件语句中“选择值”与其值相同的语句。当所有的条件语句的“选择值”与表达式的值不同时，则执行“default”后的语句。当选择值涵盖了表达式的全部结果时（如果表达式是 3 位二进制数，而选择值有 8 个），default 语句可以不要，不满足上述条件时，default 语句不可缺省。

case 语句多用于数字系统中的译码器、数据选择器、状态机及微处理器的指令译码器等电路的描述。

**【例 4.7】**用 case 语句描述 4 选 1 数据选择器。

4 选 1 数据选择器的逻辑符号如图 4.7 所示，其逻辑功能见表 4.6。由表可知，4 选 1 数据选择器的功能是：在控制输入信号 s1 和 s2 的控制下，使输入数据信号 a、b、c、d 中的一个被选中传送到输出 y。s1 和 s2 有 4 种组合值，可以用 case 语句实现其功能。

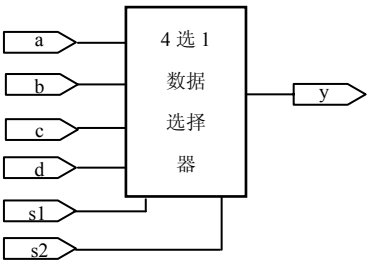


图 4.7　4 选 1 数据选择器的逻辑符号

**表 4.6　4 选 1 数据选择器逻辑功能表**

| s1　s2 | y |
|---|---|
| 0　0 | a |
| 0　1 | b |
| 1　0 | c |
| 1　1 | d |

4 选 1 数据选择器 Verilog HDL 的源程序 mux4_1 如下：

```
module      mux4_1(y,a,b,c,d,s1,s2);
input       s1,s2;
input       a,b,c,d;
output      y;
reg         y;
always  @(s1 or s2)
    begin
        case ({s1,s2})
                'b00:   y=a;
                'b01:   y=b;
                'b10:   y=c;
                'b11:   y=d;
         endcase
    end
endmodule
```

case 语句还有两种变体语句形式，即 casez 语句和 casex 语句。casez 语句和 casex 语句与 case 语句的格式完全相同，它们的区别是：在 casez 语句中，如果分支表达式某些位的值为高阻 z，那么对这些位的比较就不予以考虑，只关注其他位的比较结果；在 casex 语句中，把不予以考虑的位扩展到未知 x，即不考虑值为高阻 z 和未知 x 的那些位，只关注其他位的比较结果。

### 4.3.3 循环语句

循环语句包含 for 语句、repeat 语句、while 语句和 forever 语句 4 种。

#### 1．for 语句

for 语句的语法格式为：

```
for (索引变量 = 初值; 索引变量< 终值; 索引变量=索引变量+ 步长值)
  begin
    语句;
  end
```

for 语句可以使一组语句重复执行，语句中的索引变量、初值、终值和步长值是循环语句定义的参数，这些参数一般属于整型变量或常量。语句重复执行的次数由语句中的参数确定，即：

循环重复次数=（终值-初值）/步长值

**【例 4.8】**8 位奇偶校验器的设计。

本例用 a 表示输入信号，它是一个长度为 8 位的向量。在程序中，用 for 语句对 a 的值逐位进行模 2 加（即异或 XOR）运算，索引变量 n 控制模 2 加的次数。索引变量的初值为 0，终值为 8，因此，控制循环共执行了 8 次。8 位奇偶校验器的 Verilog HDL 源程序 ldd_8.v 如下：

```
module  ldd_8(a,out);
input[7:0]  a;
output      out;
reg         out;
integer     n;       //定义整型索引变量
always  @(a)
```

```
    begin
            out = 0;
            for(n=0; n<8; n=n+1)out=out^a[n];
    end
endmodule
```

对于一个具体的电路，可以有多种描述方法。例如，8 位奇偶校验器可以用缩减异或运算来实现，这种设计结果非常简单，其源程序 ldd_8_1.v 如下：

```
module  ldd_8_1(a,out);
input[7:0]  a;
output      out;
assign  out = ^ a;
endmodule
```

### 2．repeat 语句

repeat 语句的语法格式为：

```
repeat (循环次数表达式) 语句;
```

用 repeat 语句实现例 4.8（8 位奇偶校验器）的描述如下：

```
module  ldd_8_2(a,out);
parameter   size = 7;
input[7:0]  a;
output      out;
reg         out;
integer     n;
always  @(a)
    begin
           out = 0;
           n = 0;
           repeat(size)
              begin
                  out =out ^ a[n];
                  n = n+1;
              end
     end
endmodule
```

**注意：有的 EDA 工具软件不支持 repeat 语句，因此将 repeat 视为非法语句。**

### 3．while 语句

while 语句的语法格式为：

```
while (循环执行条件表达式)
  begin
    重复执行语句;
     修改循环条件语句;
  end
```

while 语句在执行时，首先判断循环执行条件表达式是否为真。若为真，则执行其后的语句；若为假，则不执行（表示循环结束）。为了使 while 语句能够结束，在循环执行的语句中必须包含一条能改变循环条件的语句。

### 4．forever 语句

forever 语句的语法格式为：

```
forever
  begin
    语句;
    end
```

forever 是一种无穷循环控制语句，它不断地执行其后的语句或语句块，永远不会结束。forever 语句常用来产生周期性的波形，作为仿真激励信号。例如，产生时钟时期为 20 个延迟单位（纳秒）、占空比为 50%的时钟脉冲 clk 的语句为：

```
forever #10 clk = !clk;
```

### 4.3.4 结构声明语句

Verilog HDL 的任何过程模块都是放在结构声明语句中，结构声明语句包括 4 种结构：always、initial、task 和 function。

#### 1．always 块语句

在一个 Verilog HDL 模块（module）中，always 块语句的使用次数是不受限制的，块内的语句也是不断重复执行的。always 块语句的语法结构为：

```
always @(敏感信号表达式)
  begin
    //过程赋值语句;
    // if 语句，case 语句;
    // for 语句，while 语句，repeat 语句;
    // task 语句，function 语句;
end
```

在 always 块语句中，敏感信号表达式（event-expression）应该列出影响块内取值的所有变量（一般指设计电路的输入变量或其他结构声明语句中的 reg 型变量），多个变量之间用“or”连接（也可以用逗号“,”分隔）。当表达式中任何变量发生变化时，就会执行一遍块内的语句。块内语句可以包括：过程赋值、if、case、for、while、repeat、task 和 function 等语句。always 块语句的使用，已在例 4.5 中给出。

在进行时序逻辑电路的描述中，敏感信号表达式中经常使用“posedge”和“negedge”这两个关键字来声明事件是由时钟的正边沿（上升沿）或负边沿（下降沿）触发的。若 clk 是设计电路的时钟信号，则“always @( posedge clk)”表示模块的事件是由 clk 的上升沿触发的；而“always @( negedge clk)”表示模块的事件是由 clk 的下降沿触发的。在 8 位二进制加法计数器（见例 4.2）的模块中，就使用了这类语句。

#### 2．initial 语句

initial 语句的语法格式为：

```
initial
  begin
    语句 1;
    语句 2;
……;
  end
```

initial 语句的使用次数也是不受限制的，其特点与 always 块语句相同，不同之处在于其块内的语句仅执行一次（不重复），因此 initial 语句常用于设计电路的初始化数据设置和仿真中。

### 3．task 语句

在 Verilog HDL 模块中，task 语句用来定义任务。任务类似高级语言中的子程序，用来单独完成某项具体任务，并可以被模块或其他任务调用。利用任务可以把一个大的程序模块分解成为若干小的任务，使程序清晰易懂，而且便于调试。

可以被调用的任务必须事先用 task 语句定义，定义格式如下：

```
task  任务名;
    端口声明语句;
    类型声明语句;
  begin
    语句;
  end
endtask
```

任务定义与模块（module）定义的格式相同，区别在于：任务用 task-endtask 语句来定义，而且没有端口名列表。例如，8 位加法器任务的定义如下：

```
task adder8;
output[7:0]     sum;
output          cout;
input[7:0]      ina,inb;
input           cin;
{cout,sum}=ina+inb+cin;
endtask
```

任务调用的格式如下：

```
任务名 (端口名列表);
```

例如，8 位加法器任务调用语句如下：

```
adder8 (tsum,tcout,tina,tinb,tcin);
```

完整的 8 位加法器任务调用的源程序如下：

```
module adder_8(a,b,cin,sum,cout);
input[7:0]  a,b;
input       cin;
output[7:0] sum;
output      cout;
always
    begin
        adder8(a,b,cin,sum,cout);
    end

task adder8;
input[7:0]  a,b;
input       cin;
output[7:0] sum;
output      cout;
    begin
        {cout,sum}=a+b+cin;
    end
endtask
endmodule
```